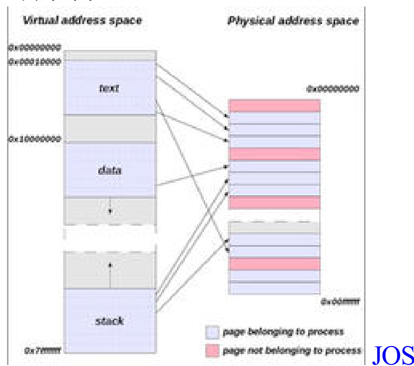
MIT 操作系统实验 MIT JOS lab1

原创

JasonLeaster 于 2014-10-03 23:03:39 发布 17431 收藏 12
分类专栏： Operating System JOS 文章标签： mit 操作系统
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/cinmyheart/article/details/39754269
版权

Operating System 同时被 2 个专栏收录
29 篇文章 3 订阅
订阅专栏



JOS
19 篇文章 11 订阅
订阅专栏

JOS lab1

首先向MIT还有K&R致敬！

没有很好的开源环境我不可能拿到这么好的东西.

向每一个与我一起交流讨论的programmer致谢！没有道友一起死磕，我也可能会中途放弃.

跟丫死磕到底.（其实这个过程会学到很多东西，很好玩很好玩，不要被panic吓到，等你都能定位panic，并修复触发panic的bug的时候，我相信大家debug的能力会上升一个水平，互勉～）

安全带系好，开始6.828号星系漫游　:)

首先先看一下MIT的课程实验lab1的安排

sep 3
LEC 1: Operating systems
(handouts: xv6 source, xv6 book)
Preparation: Unix intro
Assignment: HW: shell
Assignment: Lab 1: C, Assembly, Tools, and Bootstrapping

这里要求熟悉一下Unix的历史.

Assignment : HW: shell

In-class assignment: shell

This assignment will make you more familiar with the Unix system call interface and the shell by implementing several features in a small shell. You can do this assignment on any operating system that supports the Unix API (a Linux Athena machine, your laptop with Linux or MacOS, etc.).

Download the skeleton of the xv6 shell, and look it over. The skeleton shell contains two main parts: parsing shell commands and implementing them. The parser recognizes only simple shell commands such as the following:

```
ls > y
cat < y | sort | uniq | wc > y1
cat y1
rm y1
ls |  sort | uniq | wc
rm y
```

Cut and paste these commands into a file `t.sh` You can compile the skeleton shell as follows:

```
$ gcc sh.c
```

which produce an `a.out` file, which you can run:

```
$ ./a.out < t.sh
```

This execution will panic because you have not implemented several features. In the rest of this assignment you will implement those features.

戳下面的链接吧，具体代码去github看, 本来这贴就比较长了 ....

https://github.com/jasonleaster/MIT_6_828_assignments_2012/blob/master/sh.c

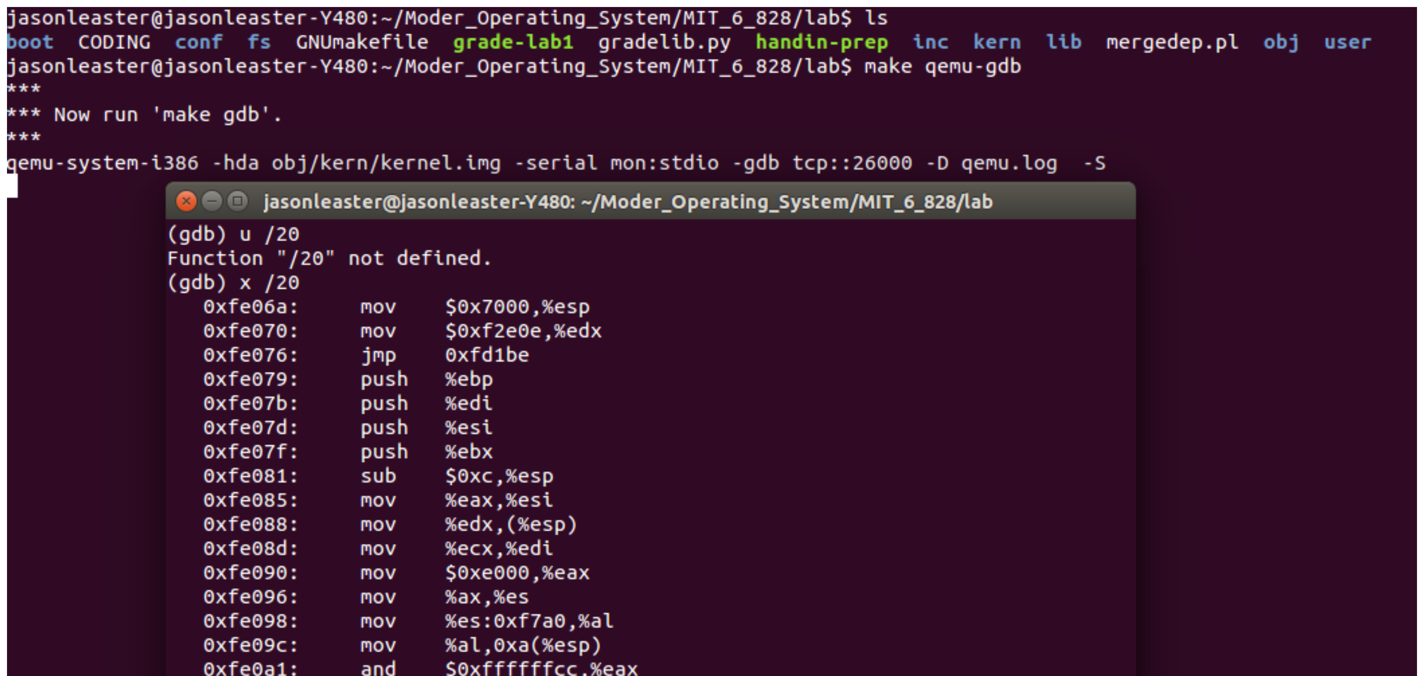
下面是单独开的shell实现分析贴:

http://blog.csdn.net/cinmyheart/article/details/45122619

Part 1: PC Bootstrap

这一部分就是很简单的介绍怎么使用qemu和gdb联调kernel...

打开两个terminal，都进入到lab目录，然后其中一个输入make qemu-gdb 另一个输入make gdb,即可看到下面的画面




```
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
```

http://blog.csdn.net/cinmyheart

注意，这里模拟的是intle 8086. 当系统复位（上电）的时候，

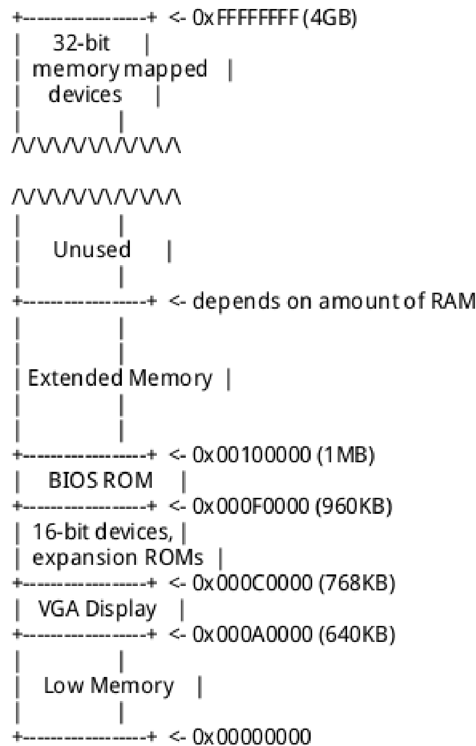
可以发现,开机后第一条指令,当前地址是0xFFFF0. 在此之前,CS == 0xFFFF

更加详细的系统 " 启动刹那间 " 分析戳下面的link:
http://blog.csdn.net/cinmyheart/article/details/42064253

有意思的是在读取BIOS信息这个阶段由于系统还有设置堆栈，gdb调试的时候step和next指令都是不能用的(需要堆栈信息),只有单行执行汇编指令的stepi指令可用，并提示一个??()的信息，当前被执行指令不在任何函数内部

The PC's Physical Address Space

We will now dive into a bit more detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:



Exercise 1. Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

第一个 exercise没有什么，只是熟悉汇编就好，都不要很牛的汇编，只要能看懂就行了．自己动手写的也不会多．

早期的intel 16bit的8086 等处理器都是只有1M的地址空间的．．．

The first PCs, which were based on the 16bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an early PC would therefore start at 0x00000000 but end at 0x000FFFFF instead of 0xFFFFFFFF.

后面对于内存的需求增大了，才把内存扩展，并为了之前的机器．就把扩展内存从0x100000开始．

The PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software.

> **Exercise 2.** Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

熟悉gdb的si指令，没话说．．．

Part 2: The Boot Loader

> **Exercise 3.** Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.
>
> Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.
>
> Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

当读取完BIOS的信息之后，这个时候就开始执行kernel的代码了

会长跳转到0x7C00地址处

When the BIOS finds a bootable floppy or hard disk, it loads the 512byte boot sector into memory at physical addresses 0x7c00 through 0x7dff, and then uses a jmp instruction to set the CS:IP to 0000:7c00 , passing control to the boot loader.

```
0x000f1067 in ?? ()
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
The target architecture is assumed to be i8086
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x /20
   0x7c01:      cld
   0x7c02:      xor    %ax,%ax
   0x7c04:      mov    %ax,%ds
   0x7c06:      mov    %ax,%es
```

从real mode切换到protected model，地址长度从16bits变为32bits！观察gdb的那个【0:7c2d】到0x7c32这种地址的表现形式我们也可以觉察到这一点

```
(gdb)
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
(gdb)
```

而后便是设置protected model下的数据段代码段等信息，然后跳转到bootmain.注意，跳转bootmain之前就设置了堆栈！

movl $start %esp

这是我们看到的最早的内核栈

```
  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain
```

这部分需要回答一部分问题:

Be able to answer the following questions:

At what point does the processor start executing 32bit code? What exactly causes the switch from 16 to 32bit mode?

从real model跳转到protected model的时候开始执行32bit code

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

boot loader最后一行代码:

```
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
```

Where is the first instruction of the kernel?

首先得定位到上面ELFHDR->e_entry指向的位置,而ELFHDR是指向0x10000(被强制类型转换成struct Elf)

```
    // read 1st page off disk
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
```

这里通过readseg使得ELFHDR得以初始化.这个初始化的数据来源就是硬盘上的内核镜像.

于是我们从那里去找这个ELFHDR->e_entry指向的位置呢？反汇编kernel镜像！

objdump -x ./obj/kern/kernel

```
体系结构: i386, 标志 0x0000011
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0010000c
```

会看到kernel的起始地址是0x10000c

设置断点就会发现这里kernel的第一条语句是

movw $0x1234, 0x472

```
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
=> 0x10000c:    movw   $0x1234,0x472
```

我们能够在 kern/entry.S中得到印证，能够找到这句代码

```
.globl entry
entry:
        movw    $0x1234,0x472
```

而kernel镜像中的entry 符号就是指向entry.S 这个文件的代码起始地址的

反汇编你会看到一个entry的符号！value是0xf010000c 这就是我们镜像上内核的入口地址了，和上面的0x10000c并不冲突，前者0x10000c是后者0xF010000C转换而来的

```
00000000 l    df *ABS*  00000000 string.c
00000000 l    df *ABS*  00000000
f010000c g       .text  00000000 entry
f0101327 g     F .text  00000020 strcpy
f010050e g     F .text  00000012 kbd_intr
f010078f g     F .text  0000000a mon_backtrace
```

这种转换一开始是手动的，我找了09 年和10年的同样的实验代码。

以前的代码(左边)　　　　　　　　　　　　现在的代码(右边)

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))();
```

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

发现这里是有手动的&转换的，而我现在用的2014年的代码是没有这种强制转换的，为这个问题纠结好久...

Many machines don't have any physical memory at address 0xf0100000, so we can't count on being
able to store the kernel there. Instead, we will use the processor's memory management hardware to
map virtual address 0xf0100000 (the link address at which the kernel code expects to run) to physical
address 0x00100000 (where the boot loader loaded the kernel into physical memory). This way, although
the kernel's virtual address is high enough to leave plenty of address space for user processes, it will
be loaded in physical memory at the 1MB point in the PC's RAM, just above the BIOS ROM. This approach
requires that the PC have at least a few megabytes of physical memory (so that physical address 0x00100000
works), but this is likely to be true of any PC built after about 1990.

因为硬件已经把0xf0100000 映射到0x100000 ，0xf010000c同理映射到0x10000c，...实质上就是手动转换变成硬件直接转换(感觉更晦涩了啊~还是手动转换的好...折腾了我一个小时)

从启动信息我们也可以知道这点(之前这个message被我无视了)

```
K> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101907 (virt)  00101907 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112944 (virt)  00112944 (phys)
Kernel executable memory footprint: 75KB
```

后来有发现自己巨渣...原来objdump的时候也可以看到信息...只怪自己弱，布吉岛啊...

这里的VMA== virtual memory address LMA == load memory address

So, 0xf0100000是虚拟地址,真正加载的时候使用的LMA，物理地址

```
Size      VMA       LMA       File off  Algn
00001907  f0100000  00100000  00001000  2**4
CONTENTS, ALLOC, LOAD, READONLY, CODE
00000730  f0101920  00101920  00002920  2**5
CONTENTS, ALLOC, LOAD, READONLY, DATA
00003871  f0102050  00102050  00003050  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
000018ba  f01058c1  001058c1  000068c1  2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
0000a300  f0108000  00108000  00009000  2**12
CONTENTS, ALLOC, LOAD, DATA
00000644  f0112300  00112300  00013300  2**5
ALLOC
00000024  00000000  00000000  00013300  2**0
CONTENTS, READONLY
```

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

根据elf格式文件储存的信息确定并读取的.

答案是:

看这值得注意的是 VMA 和LMA

Take particular note of the "VMA" (or link address) and the "LMA" (or load address) of the .text section. The load address of a section is the memory address at which that section should be loaded into memory.

The link address of a section is the memory address from which the section expects to execute. The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for.

下面是kern/kernel 的 ELF header

```
程序头:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x0000717b memsz 0x0000717b flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000a300 memsz 0x0000a944 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx

节:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001907  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000730  f0101920  00101920  00002920  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003871  f0102050  00102050  00003050  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018ba  f01058c1  001058c1  000068c1  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000644  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      00000024  00000000  00000000  00013300  2**0
```

下面是 obj/boot/boot.out的 ELF header

```
ce_code/lab$ objdump -h obj/boot/boot.out

obj/boot/boot.out:      文件格式 elf32-i386

节:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000017c  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007d7c  00007d7c  000001f0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000007b0  00000000  00000000  000002a0  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      00000846  00000000  00000000  00000a50  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      00000024  00000000  00000000  00001296  2**0
                  CONTENTS, READONLY
```

会注意到这里有些段有个 LOAD标记(比方说 .text .eh_frame)，有些没有比方说 .comment

update:2014.10.10 这里删除了我之前错误的答案，这里可能会有疑惑，等到lab2把kernel的内存分布都搞明白就知道

为什么了

Back in boot/main.c, the ph->p_pa field of each program header contains the segment's destination physical address (in this case, it really is a physical address, though the ELF specification is vague on
the actual meaning of this field).

下面这部分就是把各种 ELF header读入到内存中的过程．然后把 ELFHDR->e_entry作为函数入口

```
// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C (e.g., A tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

这个exercise 4就是提醒来踩坑的娃，童鞋哇，玩不花指针还是不要玩JOS了,好好把K&R看看再来勇敢的踩坑....

下面是这个 pointer.c的测试，如果你不debug，人脑compile然后能判断正确，基本的指针操作就差不多了

http://blog.csdn.net/cinmyheart/article/details/39755621

Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in boot/Makefrag to something wrong, run **make clean**, recompile the lab with **make**, and trace into the boot loader again to see what happens. Don't forget to change the link address back and **make clean** again afterward!

开始做＂邪恶＂的事情了．因为之前各种精心准备的链接信息是很重要的（废话）．如果链接地址不正确，程序就会出问题．这里我们就尝试改动boot/Makefrag里面的链接地址,然后试试看JOS会不会炸掉哈哈哈

并不是让我们跑去改这个Makefile而是去改链接信息，在kernel.ld里面

```
    /* AT(...) gives the load address of this section,
ells
       the boot loader where to load the kernel in phy
nory */
    /*.text : AT(0x100000) {*/
    .text : AT(0xF00000) {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
```

会注意到，我把文本段的地址改成了0xF0000，重新编译内核，然后是无法正常启动的，会挂在读取内核的那个地方，无法正常读取内核，于是就重启啊重启．这里就不上图了．

> **Exercise 6.** We can examine memory using GDB's **x** command. The GDB manual has full details, but for now, it is enough to know that the command ***x/Nx ADDR*** prints *N* words of memory at *ADDR*. (Note that both 'x's in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).
>
> Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

问的查看 BIOS enters the boot loader时候地址 0x00100000开始的八个words是什么东东

和　enter kernel的时候这个地址八个words他们之间有什么不同？

前者实际的enter boot loader地址是 0x7c00 后者的enter kernel 地址是0x10000c

我特意操作了几次，操作的意图在截图里面很明显：)

```
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x /8w 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw    $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x /8w 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb)
```

可以发现前后两次，这个地址里储存的数据是不一样的，前者是空的，后面的有些数据看不懂？没关系，我们把它当做汇编指令来看看，并把它和内核代码做一下比较看看.

一切尽在不言中！！右边是 obj/kern/kernel的汇编代码. 左边是是我们enter kernel point断点处查看的0x100000的内容

```
with this command or "print".
(gdb) x /i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
(gdb) x /20i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
   0x10000d:    movl   $0xb81234,0x472
   0x100017:    add    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)
   0x10001b:    and    %al,%bl
   0x10001d:    mov    %cr0,%eax
   0x100020:    or     $0x80010001,%eax
   0x100025:    mov    %eax,%cr0
   0x100028:    mov    $0xf010002f,%eax
   0x10002d:    jmp    *%eax
   0x10002f:    mov    $0x0,%ebp
   0x100034:    mov    $0xf0110000,%esp
   0x100039:    call   0x10009d
   0x10003e:    jmp    0x10003e
   0x100040:    push   %ebp
   0x100041:    mov    %esp,%ebp
```

```
obj/kern/kernel:     文件格式 elf32-i386


Disassembly of section .text:

f0100000 <_start+0xeffffff4>:
.globl      _start
_start = RELOC(entry)

.globl entry
entry:
    movw    $0x1234,0x472             # warm boot
f0100000:   02 b0 ad 1b 00 00         add    0x1bad(%eax),%dh
f0100006:   00 00                     add    %al,(%eax)
f0100008:   fe 4f 52                  decb   0x52(%edi)
f010000b:   e4 66                     in     $0x66,%al
```

验证了内核代码（不是bootloader）从0x100000开始，和链接脚本 kern/kernel.ld描述的一致，也和各种 ELF header描述一致 :)

Part 3: The Kernel

> **Exercise 7.** Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the **stepi** GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.
>
> What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in kern/entry.S, trace into it, and see if you were right.

不截图了，si单步调试到 movl %eax, %cr0，记得前后都要查看两个地址的内容，你会发现，在这条指令之前，两个地址的内容是不一样的. 之后就变一样了. 原因就是之前还没有建立分页机制，高地址内核区域还没有映射到内核的物理地址，而只有低地址有效的. 开启分页之后，由于有静态映射表的存在(kern/enterpgdir.c)，两块虚拟地址都指向同一块物理地址区域

主要是添加一些代码.

先把下面列出来的代码读一次

Read through kern/printf.c , lib/printfmt.c , and kern/console.c (反正我是边做边读的...)

> **Exercise 8.** We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

"We have omitted a small fragment of code the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment."

找到printfmt.c然后添加如下代码即可:

```
// (unsigned) octal
case 'o':
        // Replace this with your code.

        /*
        ** what I added. -- by EOF
        */

        num = getuint(&ap,lflag);
        base = 8;
        goto number;
```

这里因为很多机制都很健全，只要仿照着16进制输出的做一个8进制输出的初步处理就可以了

Be able to answer the following questions:

1. Explain the interface between printf.c and console.c . Specifically, what function does console.c export? How is this function used by printf.c ?

```
static void
putch(int ch, int *cnt)
{
	cputchar(ch);
	*cnt++;
}

int
vcprintf(const char *fmt, va_list ap)
{
	int cnt = 0;

	vprintfmt((void*)putch, &cnt, fmt, ap);
	return cnt;
}

int
cprintf(const char *fmt, ...)
{
	va_list ap;
	int cnt;

	va_start(ap, fmt);
	cnt = vcprintf(fmt, ap);
	va_end(ap);

	return cnt;
}
```

这里主要是说明所有的printf相关函数(JOS中),实质上都是“一层外壳”，它调用了console.c里面的putch函数.

再者,printf的实现利用到了参数变长的技巧

对于这种技巧的使用，我在这里有详细的说明:http://blog.csdn.net/cinmyheart/article/details/24582895

2. Explain the following from console.c :

```
	// What is the purpose of this?
	if (crt_pos >= CRT_SIZE) {
		int i;

		memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS)
* sizeof(uint16_t));
		for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
			crt_buf[i] = 0x0700 | ' ';
		crt_pos -= CRT_COLS;
	}
```

主要是检测当前屏幕的输出buffer是否满了，这里注意memmove其实就是把第二个参数指向的地址移动n byte到第一个参数指向的地址，这里n byte由第三个参数指定.

如果buffer满了,把屏幕第一行覆盖掉逐行上移，空出最后一行，并由for循环填充以‘ ’(空格)，最后把crt_pos置于最后一行的行首！

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

Trace the execution of the following code step by step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

In the call to cprintf() , to what does fmt point? To what does ap point?

fmt指向格式说明符字符串.ap 指向一个va_list 类型变量

不过这个代码在哪儿？我始终没有找到...以后找到update.

List (in order of execution) each call to cons_putc , va_arg , and vcprintf . For cons_putc , list its argument as well. For va_arg , list what ap points to before and after the call. For vcprintf list the values of its two arguments.

4. Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

What is the output? Explain how this output is arrived at in the stepbystep manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

会输出He110 World

我只想说...呵呵...原理嘛，就是很简单的根据ascii输出就是了

只是注意一下这里的%s部分是打印的i地址处的东东，由于是little endian机器，所以i的值在储存的时候是72 6c 64 00顺序储存的.这样对应的ascii码就是 r l d

The output depends on that fact that the x86 is littleendian. If the x86 were instead bigendian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?
Here's a description of little and bigendian and a more whimsical description.

如果是big endian嘛就是i = 0x726c6400,不需要改变57616.

5. In the following code, what is going to be printed after 'y=' ? (note: the answer is not a specific value.) Why does this happen?
cprintf("x=%d y=%d", 3);

y后会打印垃圾值

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

还是要先看变长参数的实现

```
#ifndef _STDARG_H
#define _STDARG_H

typedef char *va_list;

/* Amount of space required in an argument list for an arg of type TYPE.
   TYPE may alternatively be an expression whose type is used.  */

#define __va_rounded_size(TYPE)  \
  (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))

#ifndef __sparc__
#define va_start(AP, LASTARG)       \
 (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
#else
#define va_start(AP, LASTARG)       \
 (__builtin_saveregs (),      \
  AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
#endif

void va_end (va_list);  /* Defined in gnulib */
#define va_end(AP)

#define va_arg(AP, TYPE)      \
 (AP += __va_rounded_size (TYPE),     \
  *((TYPE *) (AP - __va_rounded_size (TYPE))))

#endif /* _STDARG_H */
```

从上面可以看到, va arg 每次是以地址往后增长取出下一参数变量的地址的。而这个实现方式就默认假 设了编译器是以从右往左的顺序将参数入栈的. 因为栈是以从高往低的方向增长的。后压栈的参数放在了内存地址的低位置,所以如果要以从左到右的顺序依次取出每个变量,那么 编译器必须以相反的顺序即从右往左将参数压栈。 如果编译器更改了压栈的顺序,那么为了仍然能正确取出所有的参数, 那么需要修改上面代码中的 va_start 和 va_arg 两个宏,将其改成用减法 得到新地址即可。感觉这地方也不少说，具体情况具体分析,不难

对于堆栈的认识最好还是去做CSAPP的lab 2 bomb~ 提前祝炸的开心:)

关于显示器颜色输出的问题:

观察cga_putc函数，

```
static void
cga_putc(int c)
{
        // if no attribute given, then use black on white
        if (!(c & ~0xFF))
                c |= 0x0700;
```

这里会检测c的8bit以上是否为0,如果是，那么黑白显示打印的字符。如果不是，那就是有蹊跷咯...

其实原理很简单

int c这个变量低8位控制显示的ascii码。接着8~15 bits用来控制颜色输出.

仅仅是为了说明原理，这里我没有把功能诠释的很完善, 高手有兴趣折腾的话欢迎交流~

修改./lib/printfmt.c 我对case ‘c’ 有小幅度的修改，增加了一个case ‘C’，增添了一个全局变量Color来传递显示何种颜色的信息

```
                // character
                case 'c':
                        ch = va_arg(ap,int) + Color;
                        putch(ch, putdat);
                        Color = 0;//reset the Color
                        break;

                /*
                **      This code added by EOF
                */
//---------------------------------------------------------------------
                case 'C':
                        switch(va_arg(ap,int))
                        {
                                case COLOR_RED:
                                Color = COLOR_RED<<8;
                                break;

                                case COLOR_GRN:
                                Color = COLOR_GRN<<8;
                                break;

                                default:
                                        Color = 0;

                        }

                        goto reswitch;
//--------------------------------------------------------------------
```

测试方法: 修改./kern/monitor.c这个文件

```
    cprintf("Welcome to %Cc the JOS kernel monitor!\n",COLOR_GRN,'H');
    cprintf("Type 'help' for a list of commands.\n");

    while (1) {
```

KO~！ 囧....其实我本意是想打印绿色的，但是对这里的高8bits的颜色控制不熟悉...So 。。。

```
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to H the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

实验本还有堆栈部分的练习，但是我觉得去拆炸弹更好，于是我就“节省时间”(偷懒一下)没做了...

http://blog.csdn.net/cinmyheart/article/details/39161471

对于JOS lab1 的实验解答还有诸多不完善的地方，以后再update....

> **Exercise 9.** Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

kern/entry.S 这个部分完成了启动时的boot stack．栈顶是$bootstacktop,而这个汇编的全局量在下图中可以看见

非常的明显，在bootloader程序的数据段内，而数据段紧紧跟在文本段之后，启动的boot stack就恰好在数据段开头位置对齐之后开始，然后是KSTKSIZE大小的栈，而后是栈顶．

```
    # Clear the frame pointer register (EBP)
    # so that once we get into debugging C code,
    # stack backtraces will be terminated properly.
    movl    $0x0,%ebp           # nuke frame pointer

    # Set the stack pointer
    movl    $(bootstacktop),%esp

    # now to C code
    call    i386_init

    # Should never get here, but in case we do, just spin.
spin:   jmp spin


.data
###################################################################
# boot stack
###################################################################
    .p2align    PGSHIFT     # force page alignment
    .globl      bootstack
bootstack:
    .space      KSTKSIZE
    .globl      bootstacktop
bootstacktop:
```

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run **make grade** to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

If you use read_ebp(), note that GCC may generate "optimized" code that calls read_ebp() *before* mon_backtrace()'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of mon_backtrace() and make sure the call to read_ebp() is happening after the function prologue.

在/kern/kdebug.c 里面会看到这段代码，关注下面的__STAB_BEGIN__ 那段代码

```
int
debuginfo_eip(uintptr_t addr, struct Eipdebuginf
{
        const struct Stab *stabs, *stab_end;
        const char *stabstr, *stabstr_end;
        int lfile, rfile, lfun, rfun, lline, rli

        // Initialize *info
        info->eip_file = "<unknown>";
        info->eip_line = 0;
        info->eip_fn_name = "<unknown>";
        info->eip_fn_namelen = 9;
        info->eip_fn_addr = addr;
        info->eip_fn_narg = 0;

        // Find the relevant set of stabs
        if (addr >= ULIM) {
                stabs = __STAB_BEGIN__;
                stab_end = __STAB_END__;
                stabstr = __STABSTR_BEGIN__;
                stabstr_end = __STABSTR_END__;
        } else {
```

__STAB_BEGIN__ __STAB_END__相关定义在/kern/kernel.ld里面

下面我给出了kernel.ld的主要内容

```
ENTRY(_start)

SECTIONS
{
 /* Link the kernel at this address: "." means the current address */
 . = 0xF0100000;

 /* AT(...) gives the load address of this section, which tells
    the boot loader where to load the kernel in physical memory */
 .text : AT(0x100000) {
  *(.text .stub .text.* .gnu.linkonce.t.*)
 }揭示了内核被加载到0x100000线性地址处

 PROVIDE(etext = .); /* Define the 'etext' symbol to this value */

 .rodata : {
  *(.rodata .rodata.* .gnu.linkonce.r.*)
 }

 /* Include debugging information in kernel memory */
 .stab : {
  PROVIDE(__STAB_BEGIN__ = .);//这里也定义了__STAB_BEGIN__等变量是0xF0100000
  *(.stab);
  PROVIDE(__STAB_END__ = .);
  BYTE(0)  /* Force the linker to allocate space
       for this section */
 }

 .stabstr : {
  PROVIDE(__STABSTR_BEGIN__ = .);
  *(.stabstr);
  PROVIDE(__STABSTR_END__ = .);
  BYTE(0)  /* Force the linker to allocate space
       for this section */
 }

 /* Adjust the address for the data segment to the next page */
 . = ALIGN(0x1000); //把数据段和bss段放到下一页

 /* The data segment */
 .data : {
  *(.data)
 }

 PROVIDE(edata = .);

 .bss : {
  *(.bss)
 }

 PROVIDE(end = .); //下一页的起始就是kernel代码段的结束位置

 /DISCARD/ : {
  *(.eh_frame .note.GNU-stack)
 }
}
```

```
lfile = 0;
rfile = (stab_end - stabs) - 1;
stab_binsearch(stabs, &lfile, &rfile, N_SO, addr);
```

首先要了解struct Stab是用来记录调试信息的结构体

关于struct stab我做了一个简介:http://blog.csdn.net/cinmyheart/article/details/39972701

kdebug.c的注释也讲的很清楚

```
// stab_binsearch(stabs, region_left, region_right, type, addr)
//
// Some stab types are arranged in increasing order by instruction
// address.  For example, N_FUN stabs (stab entries with n_type ==
// N_FUN), which mark functions, and N_SO stabs, which mark source files.
//
// Given an instruction address, this function finds the single stab
// entry of type 'type' that contains that address.
//
// The search takes place within the range [*region_left, *region_right].
// Thus, to search an entire set of N stabs, you might do:
//
//  left = 0;
//  right = N - 1;     /* rightmost stab */
//  stab_binsearch(stabs, &left, &right, type, addr);
//
// The search modifies *region_left and *region_right to bracket the
// 'addr'.  *region_left points to the matching stab that contains
// 'addr', and *region_right points just before the next stab.  If
// *region_left > *region_right, then 'addr' is not contained in any
// matching stab.
//
// For example, given these N_SO stabs:
//  Index  Type   Address
//  0      SO     f0100000
//  13     SO     f0100040
//  117    SO     f0100176
//  118    SO     f0100178
//  555    SO     f0100652
//  556    SO     f0100654
//  657    SO     f0100849
// this code:
//  left = 0, right = 657;
//  stab_binsearch(stabs, &left, &right, N_SO, 0xf0100184);
// will exit setting left = 118, right = 554.
//
```

这个stab_binsearch被函数debuginfo_eip调用，而这个函数就是为了填充struct Eipdebuginfo结构体而存在的。

始终记住这点，debuginfo_eip是为了填充struct Eipdebuginfo结构体那么在补充debuginfo_eip的时候就不会觉得迷失.

```
// Debug information about a particular instruction pointer
struct Eipdebuginfo {
        const char *eip_file;           // Source code filename for EIP
        int eip_line;                   // Source code linenumber for EIP

        const char *eip_fn_name;        // Name of function containing EIP
                                        //  - Note: not null terminated!
        int eip_fn_namelen;             // Length of function name
        uintptr_t eip_fn_addr;          // Address of start of function
        int eip_fn_narg;                // Number of function arguments
};

int debuginfo_eip(uintptr_t eip, struct Eipdebuginfo *info);
```

能在i386_init()里面找到这个函数被调用

```
    // Test the stack backtrace function (lab 1 only)
    test_backtrace(5);
```

简单的递归技巧.

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
    cprintf("entering test_backtrace %d\n", x);
    if (x > 0)
        test_backtrace(x-1);
    else
        mon_backtrace(0, 0, 0);
    cprintf("leaving test_backtrace %d\n", x);
}
```

这个部分几乎就是去让我们实现一个gdb 调试的时候的trace命令

能够利用栈的结构还有函数调用的特点，一步步的追溯到刚开始调用的函数(有点＂反递归＂的意思)

https://github.com/jasonleaster/MIT_JOS_2014/blob/lab1/kern/monitor.c

update: 2014.10.13

其实字符颜色控制还是比较简单的. 照着下面的编码来修改之前的COLOR_*** 的值就可以了

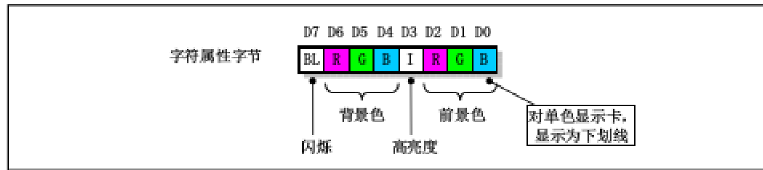


图 2-10 字符属性格式定义

与单色显示一样，图中 D7 置 1 用于让显示字符闪烁；D3 置 1 让字符高亮度显示；比特位 D6、D5、D4 和 D2、D1、D0 可以分别组合出 8 种颜色。前景色与高亮度比特位组合可以显示另外 8 种字符颜色。这些组合的颜色见表 2-5 所示。

表 2-5 前景色和背景色（左半部分）

IRGB	值	颜色名称	IRGB	值	颜色名称
0000	0x00	黑色（Black）	1000	0x08	深灰（Dark grey）
0001	0x01	蓝色（Blue）	1001	0x09	淡蓝（Light blue）
0010	0x02	绿色（Green）	1010	0x0a	淡绿（Light green）
0011	0x03	青色（Cyan）	1011	0x0b	淡青（Light cyan）
0100	0x04	红色（Red）	1100	0x0c	淡红（Light red）
0101	0x05	品红（Magenta）	1101	0x0d	淡品红（Light magenta）
0110	0x05	棕色（Brown）	1110	0x0e	黄色（Yellow）
0111	0x07	灰白（Light grey）	1111	0x0f	白色（White）

update 2015.02.13 添加了qemu的常用快捷键(话说鼠标点在qemu里面出不来了...)

组合键
Ctrl-Alt-f
全屏
Ctrl-Alt-n
切换虚拟终端'n'.标准的终端映射如下:

- n=1 : 目标系统显示
- n=2 : 临视器
- n=3 : 串口

Ctrl-Alt
抓取鼠标和键盘
Ctrl-a h
打印帮助信息
Ctrl-a x
退出模拟
Ctrl-a s
将磁盘信息保存入文件(如果为-snapshot)
Ctrl-a b
发出中断
Ctrl-a c
在控制台与监视器进行切换
Ctrl-a Ctrl-a
发送Ctrl-a

在图形模拟时,我们可以使用下面的这些组合键:

> 在虚拟控制台中,我们可以使用Ctrl-Up, Ctrl-Down, Ctrl-PageUp 和 Ctrl-PageDown在屏幕中进行移动.

在模拟时,如果我们使用`-nographic'选项,我们可以使用Ctrl-a h来得到终端命令:

update 2015.04.19

也是羞愧 ... 之前草草贴出了一些 process notes但是有些简陋，

这次更新打算重新把前面的东东强化一下，留个烙印

把没有做的challenges 做了杀一杀 好歹是第二遍了...